COMPUTING REPETITIONS IN STRINGS: CURRENT ALGORITHMS & THE COMBINATORICS OF FUTURE ONES.

COMPUTING REPETITIONS IN STRINGS: CURRENT ALGORITHMS & THE COMBINATORICS OF FUTURE ONES.

BY

EVGUENIA KOPYLOV, B.Sc.

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

AT

McMASTER UNIVERSITY

HAMILTON, ONTARIO, CANADA

DECEMBER 2010

MASTER OF SCIENCE (2010) McMaster University

(Computing & Software) Hamilton, Ontario

TITLE: COMPUTING REPETITIONS IN STRINGS: CURRENT ALGORITHMS & THE COMBINATORICS OF FUTURE ONES.

AUTHOR: Evguenia Kopylov

B.Sc., (Mathematical Science)

McMaster University, Hamilton, Canada

SUPERVISOR: Dr. William F. Smyth

NUMBER OF PAGES: xi, 47

To my family.

Abstract

Repetition is the reality and seriousness of life.

- Soren Kierkegaard

The study of repetitions exhibits roots in many modern sciences - sinusoidal waves in physics (smooth repetitive oscillations such as the electromagnetic spectrum), highly repetitive DNA in biology (tandem repeats, satellite DNA), regularities of ciphertexts in cryptography and the periodicity of sounds and sequences in music. A string on a given alphabet Σ provides the simplest common representation of this underlying property. A ***repetition*** defined on a string consists of two or more adjacent identical substrings (e.g. *abab* or *aaaa*).

A particular problem regarding repetitions is to count the number of different repetitions in a string. Conventional approaches execute in $\Theta(n \log n)$ time (Crochemore, 1981; Apostolico and Preparata, 1983; Main and Lorentz, 1984) and employ computationally heavy preprocessing. An $\Theta(n)$ time algorithm introduced in 2000 (Kolpakov and Kucherov, 2000) prevailed over its slower predecessors by succinctly encoding all repetitions as runs. A ***run*** is a maximally periodic (non-extendible) substring. For example, the string *abaabaabb* encodes 3 runs - $(aba)^2(ab)$, *aa* (twice) and *bb*. The first of these identifies three repetitions - $(aba)^2$, $(baa)^2$ and $(aab)^2$. In the early part of this thesis, we survey current algorithms for computing all repetitions.

Brute force is the essential drawback of previous attempts for detecting repetitions, despite evidence and proof that their occurrence in strings is sparse (Puglisi and Simpson, 2008). By

establishing combinatorial constraints to predict the expected sparsity of runs, extant preprocessing may be reformatted to exclude redundant computations. In (Fan *et al.*, 2006), it was shown that if two runs begin at the same position i, consequently no runs begin at some neighbouring position $i+k$. This is the fundamental idea behind our combinatorial work, in which we provide well substantiated conjectures, some of which are supported by proofs, implying that three neighbouring squares in a string force a trivial breakdown of the substring beginning at position i into repetitions of a small period.

Acknowledgements

I would like to express my uttermost gratitude to Dr. Bill Smyth for his assistance and support throughout the duration of this thesis. The final results of this research could not have been done as quickly and efficiently without his help. Köszönöm!

I would also like to thank the members of my thesis advisory committee, Dr. Jeffery Zucker and Dr. Frantisek Franek, for their helpful comments and suggestions.

Lastly, I am grateful for the kindness and support of all my colleagues and to McMaster University for providing me with this opportunity.

Notation and abbreviations

Notation. For convenience, let strings be represented in boldface (e.g. $\boldsymbol{x} = \boldsymbol{x}[0 \ldots n-1]$) and their lengths in italics (e.g. $x = |\boldsymbol{x}|$).

Contents

List of Figures

Chapter 1

Introduction

1.1 Applications

The Mozart Effect is a well-known phenomenon demonstrating the increase in mental performance and development induced by simultaneously playing two or more sound waves of certain frequencies into your ear. The superposition of the multiple sinusoidal waves arranges the resulting sound to alternate between loud and soft. Further studies to justify this anomaly have revealed that our brains may be stimulated by certain external beat frequencies to improve creativity and enhance memory (Bennet and Bennet, 2008). For example, $\beta-$waves (13-26 Hz) are best for sustaining alertness and increased analytical capabilities whereas $\theta-$waves (4-8 Hz) induce deep relaxation and have been shown to provide the best learning state (The Monroe Institute, 1985). In a recent study, the two waves were superimposed to create an optimal relaxed alertness state (Bullard, 2003). These results have spurred the development of multiple specialized software in brainwave synchronization and are excellent examples of how repetitions are applied to music and neuroscience, by means of frequencies. Specifically, a melody such as *"Mozart's Sonata for Two Pianos in D"* can be encoded into a text string by means of pitch intervals between notes or as a set of MIDI instructions (Crawford *et al.*, 1998; Cambouropoulos *et al.*, 1999). By studying the repetitions of various pitch classes found in this sonata, scientists can establish the sequence of fundamental frequencies responsible for increased brain performance.

On a slightly larger scale, repetitions are frequently observed in DNA sequences. Collectively, tandem, interspersed and single copy nucleotide repeats account for nearly half of the mammalian genome. This characteristic poses the greatest challenge in *de novo* genome assembly. Next Generation DNA sequencing techniques rely on the process of fragmenting the original DNA strands into a large collection of short sequences and then reconstructing a copy of the original chromosome via concatenation. Given the high degree of repetitive DNA, the difficulty rests in avoiding mis-assemblies associated with expansions, repeat collapses and sequence rearrangements (Schatz *et al.*, 2008). Modern assembly algorithms adopt a variety of graph manipulation methods, which are generally memory intensive and computationally difficult (NP-hard) (Pop, 2009). Studying repetitions is essential for limiting sequencing errors and deriving alternative methods for whole-genome assembly.

1.2 Historical Background

The computation of repetitions in strings has been a popular area of research since the early 1980's, with substantial focus on capturing the behavioural properties of periodicities. Early investigations into finding all unique repetitions in a string have yielded $O(n \log n)$ time algorithms (Crochemore, 1981; Apostolico and Preparata, 1983; Main and Lorentz, 1984). In 1989, repetitions were captured in a more compact form with the introduction of a run (Main, 1989). This novel idea, accompanied by earlier efforts, gave rise to an $\Theta(n)$ time algorithm for calculating all repetitions (Kolpakov and Kucherov, 2000). The structure of this algorithm consists of a chain of complementary string manipulation techniques, beginning with the suffix array construction and the LZ-factorization, followed by the computation of all left-most runs and lastly the amalgam of these constituents to determine all runs. The linearity of this algorithm depends directly on the linearity of each of its components. Given the encoding of all repetitions as runs, the maximum number $\rho(x)$ of runs in any string was shown to be

$$\rho(x) \leq k_1 x - k_2 \log_2 x\sqrt{x} \tag{1.1}$$

where k_1, k_2 are positive universal constants.

Although the size of the constants k_1, k_2 could not be inferred from the proof, computational evidence provided by Kolpakov & Kucherov supports the conjecture that the number of runs $\rho(n)$ is always less than or equal to the length of the input string, i.e. $\rho(n) \leq n$. The latest proven bounds (Simpson, 2010; Ilie *et al.*, 2008) are $0.944575712n < \rho(n) \leq 1.029n$.

LZ-factorization was initially introduced in 1976, its methods commonly applicable in lossless data compression (Lempel and Ziv, 1976). Applying this compression technique to compute regularities in strings requires the factorization of the entire string. In a recent paper (Puglisi and Simpson, 2008), it was demonstrated that repetitions in strings are sparse. More notably, the expected value $\rho(n) \leq 0.4n$ for long words ($\sigma = 2$) and $\rho(n) \leq 0.05n$ for English text ($\sigma \geq 24$). These results encouraged further research in establishing combinatorial constraints to predict the expected sparsity of runs. In (Fan *et al.*, 2006), it was shown that if two runs begin at the same position i, then necessarily no other instance of a neighbouring run may exist, suggesting $\rho(n) \not> 1n$. It may be conjectured that the next step to showing $\rho(n) \leq n$ is to establish restrictions on the number of runs that can occur near a position in a string at which one or two runs already exist, and this is the main focus of our contribution.

1.3 Contribution of This Thesis

1.3.1 Survey of algorithms for computing repetitions (Chapter 3)

We will begin by reviewing the fundamentals of existing algorithms which compute repetitions in strings. This analysis will provide the reader with an understanding of strategies historically employed to address this problem and justify our combinatorial approach.

1.3.2 Software to generate strings characterized by the NPL (Chapter 4)

The *New Periodicity Lemma* (NPL) (Fan *et al.*, 2006) states that if two squares with certain properties begin at the same position i in a string, then there cannot exist another square of certain periods at certain positions $i+k$. By relaxing the condition which restricts the existence of the neighbouring third square, we can begin to examine the periodicities and behavioural patterns associated with such a formation. There turn out to be a total of 14 possibilities for the layout of a string satisfying the existence of three squares, differing in the distance k and the lengths of the third square. The software presented here generates thousands of examples of such strings and is the foundation for the periodicity conjectures outlined in Chapter 4.

1.3.3 Periodicity conjectures generated by software (Chapter 4)

In this chapter, we present conjectures on the periodicity of a string satisfying the NPL for all 14 cases. As it happens, 7 of the 14 cases yield a fully repetitive string, whereas the remaining 7 cases vary in their output – based on data gathered from extensive computer analysis, 90% of the time the generated string is fully repetitive and the remaining 10% it consists of runs of a smaller substring.

1.3.4 The New 'Three Squares Lemma' (Chapter 5)

In this chapter, we present a new lemma for capturing the combinatorial breakdown for 7 of the 14 possible cases. The complete proofs of this lemma are provided in the forthcoming paper (Kopylov and Smyth, 2010).

Chapter 2

Terminology and Data Structures

Axel Thue (1863 - 1922) was a Norwegian mathematician and is commonly recognized as the pioneer in the theory of combinatorics on words (Berstel *et al.*, 2009). A series of his papers, (Thue, 1906, 1910, 1912, 1914), covered the earliest documented research on repetitions in strings and associated periodical properties. An ***alphabet***, defined by Σ of size σ, is a collection of letters. A finite ***string***, defined by x of length $x = n$, is a sequence of characters drawn from Σ and can be equivalently represented as $x[0 \ldots n-1]$; for $n = 1$, x is the ***empty string*** denoted as ϵ. The set of all strings over the alphabet is denoted by Σ^*. Let $u, v, w \in \Sigma^*$ and $x = uvw$, then v is a ***substring*** of x ($v \neq \epsilon$) where $u = x[0 \ldots i-1]$, $v = x[i \ldots j-1]$ and $w = x[j \ldots n-1]$ for some $0 \leq i < j \leq n-1$. Here u is a ***prefix*** of x (proper prefix if $v = w = \epsilon$) and w is a ***suffix*** of x (proper suffix if $u = v = \epsilon$). If x has both a prefix u and a suffix u, then x is said to have a ***border*** u. For example, the Fibonacci string

$$x = abaababa \tag{2.1}$$

has two borders: $b_1 = a$ and $b_2 = aba$, where b_2 is the longest border of x. If x is a concatenation of $r \geq 1$ copies of a nonempty string u, we write $x = u^r$. For $r > 1$, we say x is a ***repetition***; for $r = 2$, we say x is a ***square***; for $r = 3$, we say x is a ***cube***. For example, the string (2.1) encodes four squares – $(aba)^2$, $(ab)^2$, $(ba)^2$ and a^2. Throughout this thesis, we will assume that $x = u^r$ is a nontrivial repetition, meaning that u itself is not a repetition. If $x = u^r u'$, where u' is a prefix

of u (possibly empty), we say that x has ***period*** u. We say x is ***primitive*** if and only if its only period is x itself (i.e. $x = u$). It is easy to show that x has period u if and only if x has a border of length $x - u$ (Smyth, 2003).

Elementary components, letters or symbols, constitute the structure of a string. A prefix and a suffix are definitions for which the order of elements in a string is preserved. Fast implementations of many important string operations depend on fundamental data structures such as the suffix tree and suffix array. The following material relating to suffix trees, suffix arrays, LCP, LPF and LZ-factorization data structures closely follows that of (Al-Hafeedh *et al.*, 2010).

A *suffix tree* is a data structure used to lexicographically store all suffixes of a string. It can be computed in $O(n \log \sigma)$ time (Weiner, 1973; McCreight, 1976), where $\sigma \in O(n)$ or in $\Theta(n)$ time (Farach, 1997) if the alphabet size is known, though not practical for long strings. It is conventional to append the special character \$ to a string before constructing its suffix tree, where \$ is less than any letter in Σ. This ensures that preorder traversal of the suffix tree yields the suffixes nodes in lexicographical order. Figure 2.1 shows the suffix tree for the string $x = abaababa\$$. Note that in the suffix tree, the leaf nodes store the starting position of each suffix of x, and the inner nodes hold the lengths of the longest common prefix of all suffixes (leaf nodes) in the subtree.

A *suffix array* (SA) is a simplified alternative to a suffix tree, providing full-text indexing for a string without attending to its structural properties. Since the suffixes in a tree are in lexicographical order, traversing and listing only the leaf node values will result in a suffix array. However, this approach is computationally inefficient and more practical $\Theta(n)$ time algorithms have been devised in (Kärkkäinen and Sanders, 2003; Ko and Aluru, 2003; Nong *et al.*, 2009). Faster and more lightweight (using less space) methods have also been introduced in (Manzini and Ferragina, 2004; Maniscalco and Puglisi, 2006; Puglisi *et al.*, 2007).

A *Longest Common Prefix* (LCP) array is a data structure often supplementing the suffix array in string operations. The speedup and space efficiency of most LZ algorithms depend on both arrays

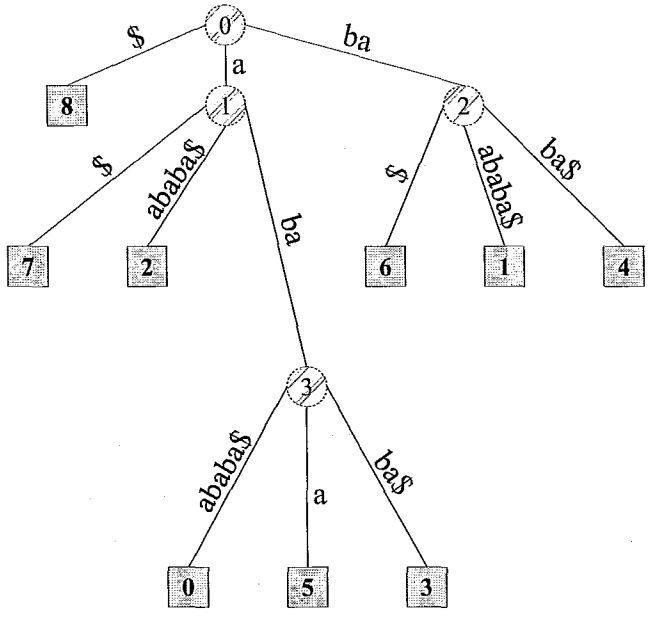

Figure 2.1: The suffix tree of $x = abaababa\$$.

to compute the LZ-factorization. The LCP array is constructed by storing the lengths of the longest common prefixes (lcp) between successive suffixes of SA, or

$$\mathrm{LCP}[i] = \mathrm{lcp}(x[\mathrm{SA}[i-1]\ldots n], x[\mathrm{SA}[i]\ldots n]) \text{ for } 1 < i \leq n.$$

For example, given two successive suffixes

$$x[\mathrm{SA}[3]\ldots 8] = x[5\ldots 8] = \underline{aba}\$,$$
$$x[\mathrm{SA}[4]\ldots 8] = x[0\ldots 8] = \underline{aba}ababa\$$$

from Figure 2.2, their longest common prefix is $\underline{aba}$ of length $\mathrm{LCP}[4] = 3$. Provided x and SA, LCP can be computed in $\Theta(n)$ time (Kasai *et al.*, 2001; Manzini, 2004; Puglisi and Turpin, 2008; Kärkkäinen *et al.*, 2009).

A *Longest Previous Factor* (LPF) array is defined as follows (Crochemore *et al.*, 2008). For any position $i \in x$, $\mathrm{LPF}[i]$ gives the length of the longest factor of x starting at position i that occurs previously in x. Formally, if $x[i]$ denotes the ith letter of x and $x[i\ldots j]$ is the factor $x[i]x[i+1]\ldots x[j]$, then

$$\text{LPF}[i] = \max\left(\{l | \boldsymbol{w}[i \ldots i+l-1] \text{ is a factor of } \boldsymbol{w}[0 \ldots i_l - 2]\} \cup \{0\}\right)$$

Applications of LPF can be seen in LZ-factorization and finding all repetitions in a string, using linear time and independent of the alphabet size (Crochemore and Ilie, 2008). The SA, LCP and LPF for the string $\boldsymbol{x} = abaababa\$$ are given in Figure 2.2.

i	SA[i]	x[SA[i]..n]	LCP[i]	LPF[i]
0	8	\$	0	0
1	7	$a\$$	0	0
2	2	$aababa\$$	1	1
3	5	$aba\$$	1	3
4	0	$abaababa\$$	3	2
5	3	$ababa\$$	3	3
6	6	$ba\$$	0	2
7	1	$baababa\$$	2	1
8	4	$baba\$$	2	0

Figure 2.2: SA, LCP and LPF arrays of $\boldsymbol{x} = abaababa\$$.

The collection of algorithms which compute repetitions directly execute in $O(n \log n)$ time and most use some form of suffix trees or suffix arrays. In (Crochemore, 1981), it was shown that Fibonacci strings of length n contain $\Theta(n \log n)$ repetitions, thus showing that these algorithms are asymptotically optimal. In 1989, Main proposed the idea to compactly encode all repetitions as "runs" and then provided a $\Theta(n)$ algorithm, starting from the suffix tree, for computing all maximal leftmost repetitions in a string (Main, 1989).

Definition 2.0.1 (run; maximal periodicity; leftmost (Main, 1989; Fan *et al.*, 2006)). A ***run*** or ***maximal periodicity*** is a substring in $\boldsymbol{x}$ of the form $\boldsymbol{p}^m\boldsymbol{q} = \boldsymbol{x}[i \ldots i+mp+q-1]$ with $m \geq 2$, $\boldsymbol{q}$ a proper prefix of $\boldsymbol{p}$, and no repetition of period p begins at position $i-1$ of $\boldsymbol{x}$ or ends at position $i+mp+q$. If $\boldsymbol{p}^m\boldsymbol{q}$ occurs more than once in $\boldsymbol{x}$, then the first time it occurs is called the ***leftmost*** occurrence.

Main's algorithm depends on the construction of a suffix tree and the LZ-factorization of a string $\boldsymbol{x}$. Initially introduced by Ziv and Lempel in 1977 (Lempel and Ziv, 1977), this method is the primary idea behind lossless text compression and it works by effectively decomposing a string $\boldsymbol{x}$ into a set of substrings by means of repeating occurrences. The key phrase *"repeating occurrences"* is what makes LZ suitable for computating various regularities in strings, such as runs.

Definition 2.0.2 (LZ-factorization (Smyth, 2003)). A decomposition $x = w_1 w_2 \dots w_k$ is an ***LZ − factorization*** if and only if each w_j, $j \in 1 \dots k$, is

(a) a letter that does not occur in $w_1 w_2 \dots w_{j-1}$; or otherwise

(b) the substring of greatest length that occurs at least once in $w_1 w_2 \dots w_{j-1}$.

For example, the Fibonacci string $x = abaababa$ has the LZ-factorization $x = a.b.a.aba.ba.$

LZ77 is based on a sliding window compression algorithm and constitutes the framework behind text compression software such as zip, gzip, and Stacker. The algorithm consists of:

(1) A sliding window of length N, where often $N = 4096$ or 8192 (search buffer);

(2) A long prefix that has already been factored;

(3) A short unfactored suffix F of approximately 18 letters (look-ahead buffer).

Suppose the algorithm has generated the first $j-1$ factors, such that $x = w_1 w_2 \dots w_{j-1}$. The next factor w_j is the longest prefix of F that matches an earlier substring within the window. When a new factor w_j has been determined, the sliding window moves from left-to-right by w_j positions. Figure 2.3 illustrates this method on the string $x = abaababa$ for the steps factoring w_4 to w_5 with $N = 3$ and $F = 3$.

LZ78 (Lempel and Ziv, 1978) was introduced a year later, using an explicit dictionary technique and followed by a series of variants (LZC, LZT, LZFG) with applications in *Graphics Interchange Format* (GIF) image compression and the UNIX compression program *compress*.

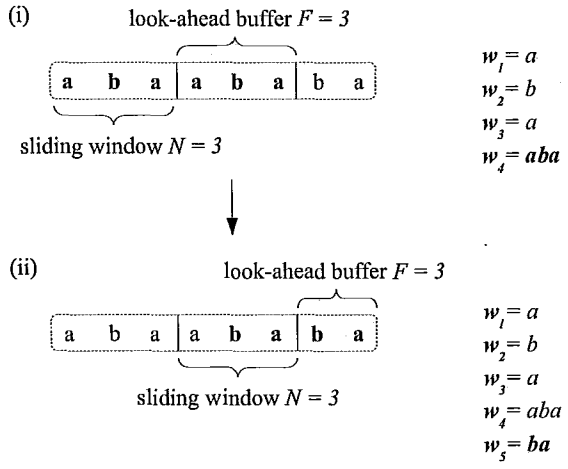

Figure 2.3: Execution of LZ77 on $x = abaababa$.

The standard of the factorization can be presented in multiple ways. In (Lempel and Ziv, 1977), the factorization is recorded as an array of three variables (POS, LEN, λ), where

POS : the location of a previous occurrence of w_j in x or the location of w_j if no previous occurrence exists;

LEN : the length (possibly zero) of the matching previous occurrence;

λ : the "letter of mismatch": for $j < k, \lambda = x[|w_1 w_2 \dots w_{j-1}| + LEN + 1]$, while for $j = k, \lambda = \$$, an arbitrary sentinel.

For the string $x = abaababa$, assuming indexing begins at $i = 0$ and $N, F = x$, the LZ77 factorization is:

$$\mathrm{LZ}(x) = 00a, 10b, 03a, 12a.$$

Most importantly, the applications of LZ-factorization can be found in lossless text compression, computing repetitions (Crochemore, 1986), maximal periodicities (Kolpakov and Kucherov, 1999; Abouelhoda *et al.*, 2004; Chen *et al.*, 2007b, 2008; Ilie *et al.*, 2008; Crochemore *et al.*, 2008) and sequence alignments (Crochemore *et al.*, 2002).

In (Kolpakov and Kucherov, 1998, 2000), it was shown that the number of runs in any string is linear in the length of the string. Along side this proof, Kolpakov and Kucherov extended Main's algorithm to include all maximal rightmost repetitions, hence computing all repetitions in time $\Theta(x)$. However, *well begun is half done*, and though the algorithm is linear in theory, its techniques are brute force and indifferent to any combinatorial reasoning. In (Fan *et al.*, 2006), it was speculated that establishing restrictions on the number of runs (squares) that can occur near a position in a string at which one or two runs already exist is the next step to reducing the so far established bounds to the "runs" conjecture $0.944575712n < \rho(n) \leq 1.029n$. The following lemma presents the first combinatorial property directed to occurrences of neighbouring squares in a string:

Lemma 2.0.3 ("The Periodicity Lemma" (Fine and Wilf, 1965)). *Let p and q be two periods of $x = x[1 \ldots n]$, and let $d = \gcd(p, q)$. If $p + q \leq n + d$, then d is also a period of x.*

However, this lemma provides no information about occurrences of runs in x and forces no restrictions on the positions of possible periodic substrings. The following 'three squares lemma' provides such information.

Lemma 2.0.4 ("Three Squares Lemma" (Crochemore and Rytter, 1995)). *Suppose u is not a repetition, and suppose $w \neq u^j$ for any $j \geq 1$. If u^2 is a prefix of w^2, in turn a proper prefix of v^2, then $w \leq v - u$.*

This lemma examines the combinatorial consequence of having three squares beginning at the same position. *The New Periodicity Lemma* (NPL) in (Fan *et al.*, 2006), is a generalization of this result. In this thesis we extend the results of (Fan *et al.*, 2006) and make them more precise. Prior to introducing the details of NPL, let us consider the following definitions:

Definition 2.0.5 (irreducible). A square u^2 is said to be ***irreducible*** if u is not a repetition.

Definition 2.0.6 (regular). A square u^2 is said to be ***regular*** if no prefix of u^2 is a square.

Lemma 2.0.7 ("The New Periodicity Lemma" (Fan *et al.*, 2006)). *If x has regular prefix u^2 and irreducible prefix v^2, $u < v < 2u$, then for every $k \in 0 \ldots v - u - 1$ and every $w \in w - u + 1 \ldots v - 1$, $w \neq u$, $x[k + 1 \ldots k + 2w]$ is not a square.*

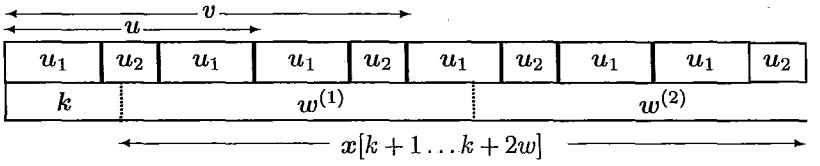

Figure 2.4: Three squares satisfying the NPL.

Without loss of generality, we may suppose that $3u/2 < v < 2u$, $v - u < w < v$ and $w \neq u$. (The case for which $u < v \leq 3u/2$ was studied in (Kopylov and Smyth, 2010)). It is shown in (Fan *et al.*, 2006) that if $\boldsymbol{u}$ is regular (and $\boldsymbol{u}$ is irreducible), then the substring $\boldsymbol{v}^2$ must have the structure

$$\boldsymbol{v}^2 = \boldsymbol{u}_1\boldsymbol{u}_2\boldsymbol{u}_1\boldsymbol{u}_1\boldsymbol{u}_2\boldsymbol{u}_1\boldsymbol{u}_2\boldsymbol{u}_1\boldsymbol{u}_1\boldsymbol{u}_2, \tag{2.2}$$

where $u = 2u_1 + u_2$ and $v = 3u_1 + 2u_2$.

It is shown in (Kopylov and Smyth, 2010) that (2.2) also holds whenever $3u/2 < v < 2u$, where it is necessary only to make the natural assumption that $\boldsymbol{u}^2$ and $\boldsymbol{v}^2$ are irreducible – that is, expressed in terms of the generator of least length. The analysis of the three neighbouring squares is distributed amongst 14 possible cases, differing in the range for $\boldsymbol{k}$ and the end positions of $\boldsymbol{w}^{(1)}$ and $\boldsymbol{w}^{(2)}$. Note that $\boldsymbol{k} + \boldsymbol{w}^2$ can extend past $\boldsymbol{v}^2$. The proof for each of the 14 cases shows the existence of a square in a prefix of $\boldsymbol{u}$ which is forced by the presence of $\boldsymbol{w}^2$, hence contradicting the regularity of $\boldsymbol{u}^2$ and concluding that if $\boldsymbol{u}^2$ is not regular then the square prefix must exist.

The material in the NPL paper was shortly followed by a question addressing whether the existence of $\boldsymbol{w}^2$ forces $\boldsymbol{u}$ to be a repetition, and hence the entire string. The approach to examine this query was to construct strings satisfying each of the cases by relaxing the regularity condition of $\boldsymbol{u}$ and varying the lengths of $\boldsymbol{u}_1$, $\boldsymbol{u}_2$, $\boldsymbol{k}$ and $\boldsymbol{w}$. These experiments have ultimately aided us in formulating conjectures for the combinatorial behaviour in strings within all 14 cases.

Chapter 3

Algorithms for Computing Repetitions and Runs

3.1 Repetitions vs. Runs

In this chapter, we will begin by briefly summarizing the original algorithms for directly computing all repetitions in a string, collectively executing in $O(n \log n)$ time. Since it was altogether shown in (Crochemore, 1981; Crochemore and Rytter, 1995) that the maximum number of a repetitions in a string of length n is $O(n \log n)$ (using Fibonacci words), these methods are asymptotically optimal. Next, we will discuss how the emergence of two linear time algorithms (Runs I, II) for finding all runs in a string allowed for indirect computation of all the repetitions in $\Theta(n)$ time. The following list shows the steps taken by both algorithms to compute all runs:

(1) *Compute LZ-factorization of* $\boldsymbol{x}$

→ $\Theta(n)$ time, provided a $\Theta(n)$ ST/SA construction algorithm is used.

(2) Runs I (Main, 1989): Find all leftmost runs in $\boldsymbol{x}$ using LZ

→ $\Theta(n)$ time.

(3) Runs II (Kolpakov and Kucherov, 1998): Find all runs in $\boldsymbol{x}$ (using the leftmost ones)

→ Proven to be executable in $\Theta(n)$ time; however the proof depends solely on the fact that the maximum number of runs $\rho(n) \in O(n)$.

The LZ-factorization of a string conforms to a sequential, multi-step process. Over time, emergence of full and succinct (compressed) data structures such as the SA, Enhanced Suffix Array (SA plus an "lcp-interval tree" (Abouelhoda *et al.*, 2004)), LPF and BWT (Burrows-Wheeler transform) arrays has significantly improved the space and time complexities as studied in (Al-Hafeedh *et al.*, 2010); however the complete procedure remains computationally demanding. Figure 3.1 presents the collection of current algorithms and their implemented data structures, applicable on any given string x. Prior to examining the details of Runs I and II, we will present the leading LZ-factorization methods available to date. In conclusion to the chapter, we will propose a new combinatorial approach to finding all runs by studying the expected breakdown of a string when two squares begin at the same position and a third one occurs nearby.

During the latter part of the 20th century, the first algorithm to compute all repetitions in a string was presented by (Crochemore, 1981). This method applied Hopcroft's $O(n \log n)$ algorithm for minimizing finite state automata (Hopcroft, 1971) to form equivalence relations through sequential refinement of the string's indices. Shortly after, an alternate off-line approach was given in (Apostolico and Preparata, 1983) with the application of suffix tree properties and a "leaf tree" data structure. Lastly, (Main and Lorentz, 1984) introduced a recursive, divide-and-conquer algorithm based on a linear procedure for finding all new repetitions formed through the concatenation of two strings. Though the difference of approach in all three cases is striking, all invariably yield an optimal time of $O(n \log n)$. As noted in the first paragraph of this chapter, it was shown by Crochemore that Fibonacci strings of length n contain $\Theta(n \log n)$ repetitions.

The solution to reduce the processing time appeared in (Main, 1989), by compactly encoding all repetitions as runs. Main's algorithm uses the LZ-factorization to compute all leftmost occurrences of distinct runs in a string. Kolpakov and Kucherov later showed that the number of runs in any string is linear in the length of the string. However, since the constant of proportionality was not defined from their proof and can therefore be arbitrarily large, the "linearity" of the algorithm

depends entirely upon $\rho(n) \in O(n)$, as shown in Theorem 3.1.1 from (Kolpakov and Kucherov, 1998). By modifying Main's algorithm, they were able to compute all runs from the leftmost ones in time proportional to all runs.

Theorem 3.1.1. *Let $\rho(n)$ be the maximum number of runs that can occur in any string of length n on any alphabet. Then there exist positive constants k_1 and k_2 independent of n such that for every integer $n \geq 1$,*

$$\rho(n) \leq k_1 n - k_2 \sqrt{n} \log_2 n. \tag{3.1}$$

As previously noted, the size of k_1 and k_2 could not be derived from the proof; however computational evidence provided in (Kolpakov and Kucherov, 1998), suggests the following:

1. $\rho(n) \leq n$;
2. $0 \leq \rho(n+1) - \rho(n) \leq 2$;

Up to now, conjecture (1) has attracted the most attention, with proven bounds (Simpson, 2010; Ilie *et al.*, 2008) $0.944575712n < \rho(n) \leq 1.029n$.

3.2 LZ-factorization

Both Main's algorithm for computing runs and the later one provided by Kolpakov and Kucherov entail the LZ-factorization of the input string. The leading challenge remains in improving the runtimes and memory storage of algorithms responsible for carrying out LZ-factorization, given that current methods depend upon complex data structures, whose computation accounts for 80% or more of the overall runtime, and are essentially brute force.

Albeit the LZ outputs differ slightly for some of the algorithms illustrated in Figure 3.1, for general purposes such as the computation of regularities, they are comparable. Explicit details of the experimental results comparing time and space requirements for these methods are given in

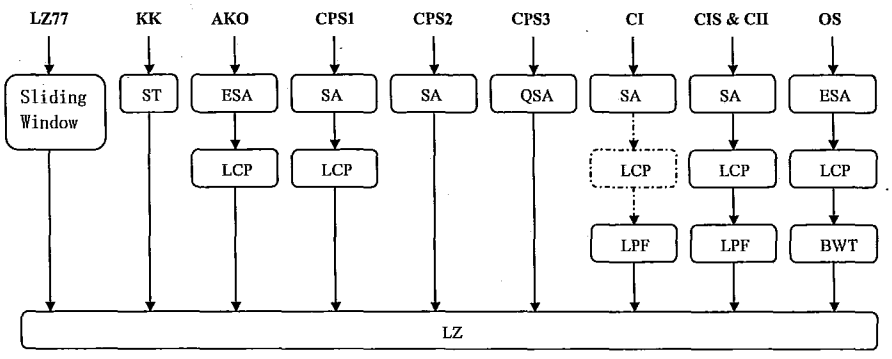

Figure 3.1: The steps and main data structures used by each algorithm

(Al-Hafeedh *et al.*, 2010). Nevertheless, it was found that all algorithms listed in Figure 3.1 were not greatly different, as shown in Figures A.1 and A.2 of the Appendix. The algorithms denoted as *CPS1b* and *CIS* outperformed their adversaries on files which were not highly periodic, with asymptotic worst-case times $\Theta(n)$ and requiring as little as $1.5n$ space. Note that *CPS1b* and *CIS* implement full data structures (SA, LCP and LPF) unlike *CPS2*, which uses a succinct data structure (SA and range minimum queries, discussed further in the chapter). Therefore, though the running time for *CPS2* is proportionately slower, the storage space is also much less. Algorithms *CPS1b*, *CPS2* and *CIS* are the three representatives that we provide some description of in this thesis. Likewise, algorithms *CPS3* and *OS* implement succinct data structures. Mainly, *OS* stores the SA in a compressed form and uses **rank/select** operations (Lee and Park, 2007) and range minimum queries (see sec. on *CPS2*) to access and update array entries; whereas *CPS3* uses a QSA structure (Franek *et al.*, 2003) and q-grams (Ukkonen, 1992) to compute LZ.

3.2.1 CPS1b

A collection of fast and space efficient algorithms known as *CPS1a*, *CPS1b* and *CPS1c*, was introduced in (Chen *et al.*, 2007b) to compute the LZ-factorization of a string. The pseudocode in Figure 3.2 is a modified version of the paper's original, specifically edited to exemplify the functionality of *CPS1b*. The algorithm begins by assigning three pointers i_1, i_2, i_3 to positions in the SA such that $0 \leq i_1 < i_2 < i_3 \leq n$. The scheme behind *CPS1b* and its close variants is based upon the following two observations:

(*i*) if $\text{LCP}[i_1] < \text{LCP}[i_1+1]$, then two or more repeating substrings with the same LCP have begun in x. Meaning, the suffixes $j = \text{SA}[i_1]$ and $j' = \text{SA}[i_1 + 1]$ have a unique longest common prefix of length $\text{LCP}[i_1 + 1]$ such that $\forall k \in [1, i_1 - 1]$ there does not exist a position $\text{SA}[k]$ sharing this prefix.

(*ii*) if $\text{LCP}[i_2] > \text{LCP}[i_2+1]$, then two or more repeating substrings with the same LCP have ended in x. Meaning, the suffixes $j = \text{SA}[i_2 - 1]$ and $j' = \text{SA}[i_2]$ have a unique longest common prefix of length $\text{LCP}[i_2]$ such that $\forall k \in [i_2 + 1, n + 1]$ there does not exist a position $\text{SA}[k]$ sharing this prefix.

Firstly, a left-to-right traversal of the SA is performed to locate the next position $i_2 < i_3$ such that $\text{LCP}[i_2] > \text{LCP}[i_3]$ (lines 3-4). Secondly, a stack S is used to backtrack from i_2 to the first position $i_1 < i_2$ such that $\text{LCP}[i_1] < \text{LCP}[i_2]$ (lines 8-12), at each step setting the larger position in POS corresponding to equal LCP to point leftwards to the smaller one (lines 18-23), until the LCP value for position i_1 popped from S falls below $\text{LCP}[i_2]$. After processing each collection of repeating substrings, the pointers i_1, i_2 and i_3 are reset for the next stage to check whether other sequences of pairs (POS, LEN) may exist at this position (lines 14-17).

In *CPS1b*, the storage for SA and LCP is reduced by n words (compared to *CPS1a*) because it is dynamically reused for specifying the location and contents of POS. In total, the space requirement for *CPS1b* is $3.25n$ words plus stack. The output of the LZ-factorization is in the form (POS, LEN), which is duly applicable for computing lossless text compression as well as runs.

Figure 3.3 shows an example of how algorithm *CPS1b* is executed on the string $x = abaababa\$$.

3.2.2 CPS2

A year later, two new methods *CPS2* and *CPS3* (Chen *et al.*, 2008) joined the family of *CPS* algorithms. This new addition favoured space over time using a succinct algorithm, with asymptotic worst-case times $O(n \log n)$ and $O(n^2)$ in contrast to the linear *CPS1* algorithms, but conversely requiring significantly less storage space, $1.25n - 1.5n$ words respectively. An important difference

```
     — Using SA_x and LCP_x, compute LEN[0...n-1].
     — Compute POS_x by in-place compactification of SA_x and LCP_x
     — into SA_x; e.g. POS[SA[i]] = LCP[i].
(1)    i1 ← 0; i2 ← 1; i3 ← 2
(2)    while i3 ≤ n do
     — Identify the next position i2 < i3 with LCP[i2] > LCP[i3].
(3)      while LCP[i2] ≤ LCP[i3] do
(4)        push(S, i1)
(5)        i1 ← i2; i2 ← i3; i3 ← i3 + 1
(6)      p1 ← SA[i1]; p2 ← SA[i2]; l2 ← LCP[i2]
(7)      assign(p1, p2, POS)
(8)      while LCP[i1] ← l2 do
(9)        i2 ← i1
(10)       i1 ← pop(S)
(11)       p1 ← SA[i1]
(12)       assign(p1, p2, POS)
(13)     SA[i1] = p2
     — Reset pointers for the next stage.
(14)     if i1 > 1 then
(15)       i2 ← i1; i1 ← pop(S)
(16)     else
(17)       i2 ← i3; i3 ← i3 + 1

(18)   procedure assign(p1, p2, POS)
(19)     if (p1 < p2) then
(20)       SA[i2] ← p2; LCP[i2] ← p1; LEN[p2] = l2
(21)       p2 ← p1
(22)     else
(23)       SA[i2] ← p1; LCP[i2] ← p2; LEN[p1] = l2
```

Figure 3.2: Algorithm CPS1b.

between algorithms *CPS1b* and *CPS2* is that the former outputs a (POS, LEN) pair for every position in the string x, whereas the latter only returns information about the positions where factors truly exist.

Rather than constructing both the SA and LCP, the algorithm *CPS2* (Figure 3.4) uses only the SA and a data structure $\mathrm{RMQ}_{\mathrm{SA}}$ for answering range minimum queries on SA (Bender and Farach-Colton, 2000; Harel and Tarjan, 1984). $\mathrm{RMQ}_{\mathrm{SA}}(lb, rb)$ provides the index of the minimum value among $\{\mathrm{SA}[lb], \mathrm{SA}[lb+1], \ldots, \mathrm{SA}[rb]\}$ in constant time and $O(n)$ space (Chen *et al.*, 2007a), and $\mathrm{SA}[\mathrm{RMQ}_{\mathrm{SA}}(lb, rb)]$ uses this index to compute the minimum $\mathrm{SA}[lb \ldots rb]$ (line 12). In the function `lzfactor`, this minimum value is repeatedly computed on the range $lb \ldots rb$, which narrows if a

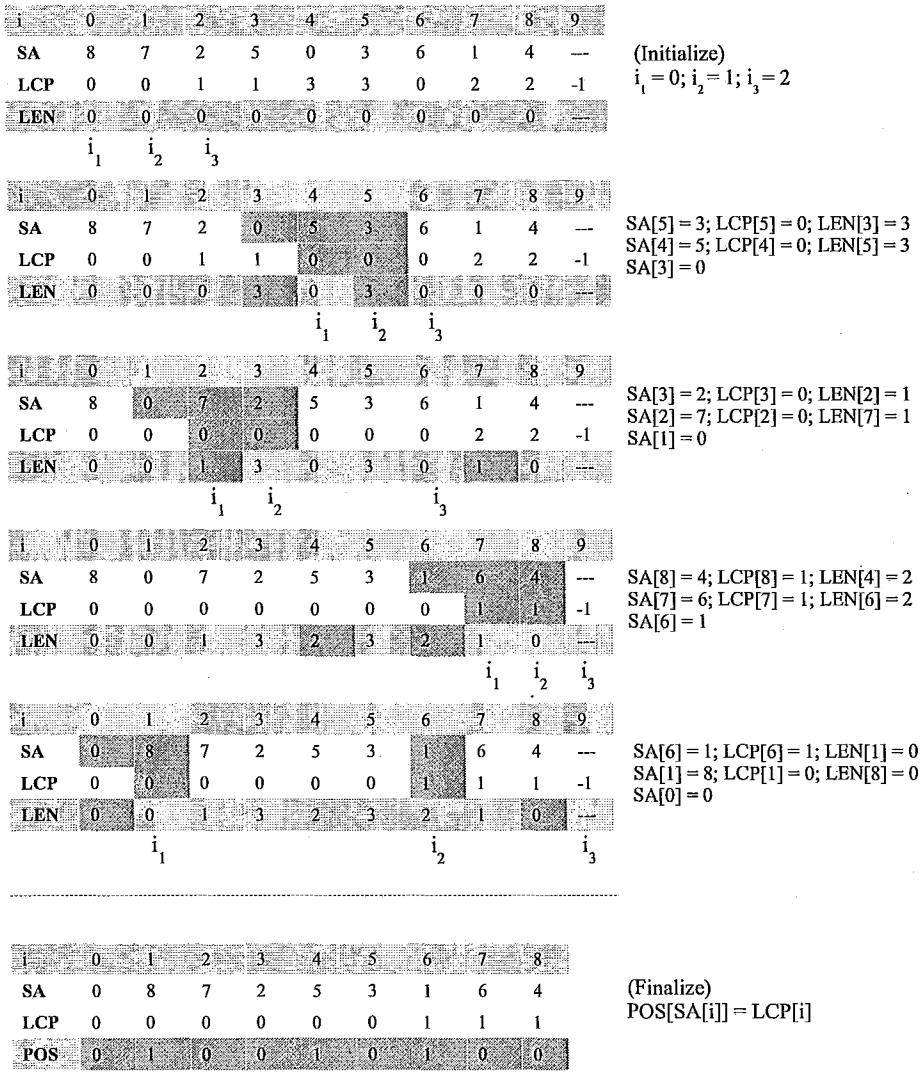

Figure 3.3: Execution of *CPS1b* on $x = abaababa\$$.

SA$[lb \ldots rb]$ begins at some position $p < i$ in $\boldsymbol{x}$, until the longest substring beginning at position i matches a previous position in $\boldsymbol{x}$ (line 15). The range $lb \ldots rb$ is calculated by the function `refine` in $O(\log n)$ time (line 11).

```
(1)   output (1,1)
(2)   i ← 2
(3)   while i ≤ n do
(4)       (POS,LEN) ← lzfactor(x, SA, i)
(5)       output (POS,LEN)
(6)       i ← i + LEN

   — Using SA_x and RMQ_SA compute the position
   — and length of the LZ factor beginning at i in x.
(7)   function lzfactor(x, SA, i)
(8)   match ← i
(9)   lb ← 1; rb ← n; j ← i
(10)  repeat
(11)      (lb, rb) ← refine(lb, rb, j − i, x[j])
(12)      min ← SA[RMQ_SA(lb, rb)]
(13)      if min < i then
(14)          match ← min; j ← j + 1
(15)  until min ≥ i or j > n
(16)  return (match, j − i)
```

Figure 3.4: Algorithm CPS2.

3.2.3 CIS

The *CIS* algorithm is similar to *CPS1b* in using full data structures, SA and LCP; however, it differs by computing the LPF array first and then LZ. The worst-case run time of this algorithm is $\Theta(n)$ and the basic idea includes computing LPF[SA$[j]$] for $j \in [0, n-1]$ immediately by means of cases (i) and (ii) defined below. To facilitate a clear understanding of this method, the SA and LCP values (Figure 2.2) for the string $\boldsymbol{x} = abaababa$ are represented graphically in Figures 3.5 and 3.6. The vertices are labeled with SA values; listed vertically in ascending order corresponding to their starting position in the string and horizontally in their original order of occurrence. The solid edges connecting any two vertices represent their associative LCP values. The dashed edges represent a new connection resulting from the qualifying case (i) or (ii). A similar example can be found in

(Crochemore *et al.*, 2008).

(*i*) if SA[j] > max(SA[$j-1$], SA[$j+1$]) (peak in the graph), then LPF[SA[j]] = max(LCP[j], LCP[$j+1$]). Referring to Figure 3.5, for $j = 4$, the value LPF[SA[4]] = LPF[5] = max(LCP[4], LCP[5]) = 3 (maximum of the labels of the 2 adjacent edges). Since SA[4] = 5, the vertex labeled 5 can be removed from the graph. An edge between vertices 0 and 3 is created, labeled by min(LCP[4], LCP[5]) = 3 (minimum of the two labels).

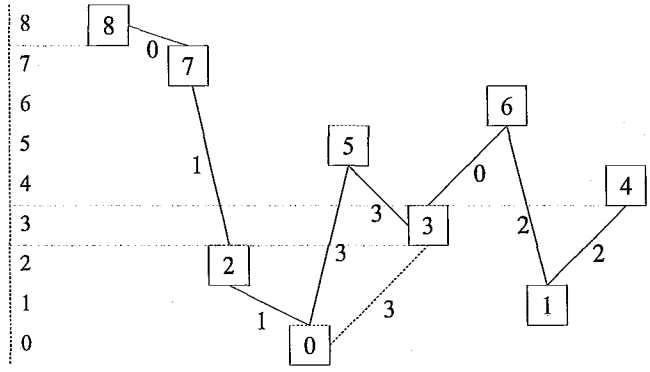

Figure 3.5: Step (*i*) for the text $x = abaababa$.

(*ii*) if SA[$j-1$] < SA[j] < SA[$j+1$] and LCP[j] ≥ LCP[$j+1$] (SA values increasing and LCP values decreasing), then LPF[SA[j]] = LCP[j] since nothing larger than LCP[j] can be obtained from LCP[$j+1$]. Assume the graph in Figure 3.5 was modified by calculations in (*i*) to form the graph in Figure 3.6. For $j = 5$, the value LPF[SA[5]] = LPF[3] = 3. Then as before, since SA[5] = 3, the vertex 3 can be removed from the graph and the vertices 0 and 6 connected by an edge labeled 0 = LCP[$j+1$] = LCP[6].

CIS applies the appropriate case from above with a left-to-right scan of the SA and stacks entries i = SA[j] that do not immediately yield an LPF value. With at most $2\sqrt{2n}$ single integer entries, the worst-case space requirement is $3n+2\sqrt{2n}$ words (Al-Hafeedh *et al.*, 2010). After the LPF array is constructed, a simple algorithm in Figure 3.7 given by (Crochemore and Ilie, 2008) computes LZ.

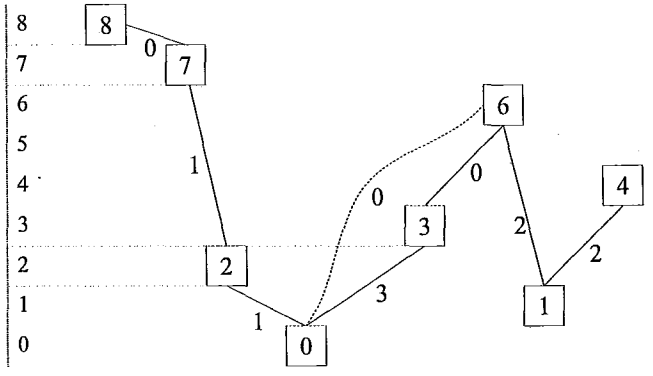

Figure 3.6: Step (ii) for the text $x = abaababa$.

$\mathrm{LZ}[0] \leftarrow 0;\ i \leftarrow 0$
while $\mathrm{LZ}[i] < n$ **do**
    $\mathrm{LZ}[i+1] \leftarrow \mathrm{LZ}[i] + \max\left(1, \mathrm{LPF}[\mathrm{LZ}[i]]\right)$
    $i \leftarrow i+1$
return LZ

Figure 3.7: Given LPF for a string x, compute LZ.

3.3 Runs I: Detecting leftmost maximal periodicities

The idea to use LZ-factorization, initially noticed in (Crochemore, 1984), was elaborated in (Main, 1989) to compute all leftmost occurrences of runs in a string in $\Theta(n)$ time. Originally, the factorization could be computed using McCreight's linear-time (for a fixed finite alphabet) suffix tree construction algorithm (McCreight, 1976). However, as demonstrated in the previous section, modern algorithms can use suffix arrays and succinct data structures to consume less storage space and output faster results.

Main used LZ-factorization as the principle to introduce two new properties for leftmost periodicities,

Theorem 3.3.1. *Let x be a string with LZ-factorization $x = w_1 \dots w_k$, and let r be a periodicity of x, with the leftmost occurrence of r at $x[i \dots j]$, and with $x_j \in w_h$ for some $h \leq k$. Then,*

(1) *x_i occurs before w_h, and*

(2) *if we let* $\boldsymbol{r} = \boldsymbol{r}_L\boldsymbol{r}_R$ *for some prefix* $\boldsymbol{r}_R$ *of* $\boldsymbol{w}_h$,

$$r_L < 2w_{h-1} + w_h.$$

These properties capture the necessary conditions in calculating all maximal leftmost periodicities, as presented in the following algorithm taken from (Smyth, 2003; Main, 1989),

```
      — Compute LZ-factorization x = w_1 ... w_k
(1)   j ← 0
(2)   for h ← 2 to k do
(3)       — Compute the maximum-length prefix r_L
(4)       — of a run ending in w_h in x (Theorem 3.3.1)
(5)     j ← j + w_{h-1}
(6)     l ← min(j, 2w_{h-1} + w_h)
(7)     r_L ← x[j − l ... j − 1]
(8)       — Compute all leftmost runs in x, ending in w_h
(9)     calcruns(r_L, w_h)
```

Figure 3.8: Main's algorithm for computing all maximal leftmost runs.

It is important to note that this algorithm does not exclusively output only maximal leftmost periodicities - runs which are not maximal or leftmost may also be found. Computation of LZ-factorization, and the immediate 3 steps following the **for** loop are linear in time. The function `calcruns` is derived from an algorithm specific to repetitions (Main and Lorentz, 1984) which finds all new squares appearing in a concatenation of two strings and executes in time $O(r_L + w_h)$. However, since $r_L < 2w_{h-1} + w_h$,

$$2n \leq \sum_{h=2}^{k}(2w_{h-1} + w_h + w_h) = \sum_{h=2}^{k} 2(w_{h-1} + w_h) < 4n. \tag{3.2}$$

Altogether, the time complexity of the algorithm in Figure 4.1 is $\Theta(n)$, given a linear time LZ construction algorithm is used.

Figure 3.9 gives an example of Main's algorithm executed on the string $\boldsymbol{x} = abaabaabb$ with $\mathrm{LZ}(\boldsymbol{x}) = a.b.a.abaab.b$. A similar example is provided in Main's paper.

h	j	$2w_{h-1}+2_h$	l	r_L	$r_L w_h$	calcruns(r_L, w_h)
2	0	3	1	$x[0\ldots0]=a$	$a.b$	0
3	1	3	2	$x[0\ldots1]=ab$	$ab.a$	0
4	2	7	3	$x[0\ldots2]=aba$	$aba.abaab$	$(aba)^2ab, (baa)^2b, (aab)^2, a^2$
5	7	11	8	$x[0\ldots7]=abaabaab$	$abaabaab.b$	b^2

Figure 3.9: Finding all leftmost runs using Main's algorithm for string $x = abaabaabb$.

3.4 Runs II: On maximal repetitions in words

Less than a decade after Main suggested his linear time algorithm to compute all leftmost distinct maximal repetitions in a string, (Kolpakov and Kucherov, 1998) published a slightly modified linear algorithm to compute *all* runs. The strategy was to notice that all the runs can be recovered from the leftmost ones using basic string manipulation techniques. For every factor w_h in the LZ factorization $w_1 w_2 \cdots w_k$ of x, they stored its start position i_h and the start position i'_h of some previous copy (zero if no previous copy exists). These positions are stored in two arrays $I = I[1..k+1]$ and $I' = I'[1..k]$, computed using the suffix tree as a byproduct of the original LZ computation.

In the algorithm, every leftmost run is stored as a pair (i, j), corresponding to the run's initial and final positions in x. Throughout the computation, an array OCCURS$[0\ldots n-1]$ is formed that stores pointers to linked lists of positions $j_{i1}, j_{i2}, \ldots, j_{ir_i}$. These positions represent the final positions of leftmost runs initially occurring at position i in x. The next step performs bucket sort on the pairs (i, j), and puts them in an increasing order based on the end position j. The array OCCURS is then updated by traversing through the sorted list, and matching the corresponding initial positions i with those values of j it points to. This method maintains the invariant $j_{i1} < j_{i2} < \cdots < j_{ir_i}$ and more importantly, performs in linear time (since the number of runs has been proven to be linearly bounded).

The last step involves finding an instance of w_h at some position $I'[h] < I[h]$ and shifting all of the maximal repetitions, ending inside w_h, $i_h - i'_h$ steps to the right. Given that OCCURS[i] holds the start positions of the runs in increasing order of length, the overall 'shifting' process is linear in time.

```
— Given the arrays I[0...k] and I'[0...k-1] that describe the
— LZ-factorization w_1 w_2 ... w_k of x[0...n-1], and given lists
— OCCURS[0...n-1] that specify in increasing order of length
— all leftmost runs that occur at each position i, this
— algorithm updates the lists to include all runs in x.

(1)  for h ← 2 to k do
(2)    if I'[h] > 0 then
(3)      δ ← I[h] − I'[h]  — the offset of w_h from its copy
(4)      for i ← I[h] to I[h+1] − 1 do
(5)        ∀j ∈ list(OCCURS[i − δ])
(6)            — a maximal rightmost run must begin and end in w_h
(7)          if (j + δ) − i < I[h+1] − i then
(8)            insert list(OCCURS[i]) ← j + δ
```

Figure 3.10: Algorithm to find all runs in x (Smyth, 2003).

3.5 Combinatorial Approach to Computing Runs

It is apparent that although the 'runs computing' algorithms presented in this chapter are $\Theta(n)$ in time, their collective procedures are essentially brute force, and do not embody any combinatorial properties such as the expected sparsity of runs. Roughly speaking, the time to complete the sum of the first n natural numbers using brute force can be costly, yet immediate with a few simple combinatorial arguments. Possible future approaches to computing runs may perhaps perform a left-to-right traversal of a string, taking into account the repetitions (squares) existing at every position. Then, if there exists a setup of 3 neighbouring squares as studied in this thesis, we can combinatorially compute the number of repetitions they include and skip the length of the breakdown to continue processing the next part of the string. Current methods perform preprocessing on every letter of the string and thus are computationally intensive in practice.

The forthcoming chapters of this thesis are devoted to our study of combinatorial effects arising from having three squares occur at neighbouring positions in a string. The conditions posed for such a layout are provided by Lemma 2.0.7. Given this structural environment, we describe the possible layout of a string x using 14 subcases and then go on to present software which generates strings corresponding to each subcase. Using the results of this software, along with those provided by an accompanying conjecture verification code, we produce conjectures on the combinatorial behaviour

in all of the 14 subcases that arise. Proofs to seven of the 14 subcases are to be published in (Kopylov and Smyth, 2010). In all of the proved cases, the assumed existence of three neighbouring squares forces a trival repetition of small period. In the remaining cases, the conjectured breakdowns are of a highly periodic form.

Chapter 4

Conjectures Related to Neighbouring Squares

4.1 New Software

In this section we will describe the algorithm used to generate strings for which two squares, u^2 and v^2, begin at the same position i and a third square w^2 occurs nearby at position $i+k$. We do not assume any conditions on u^2 or v^2 such as regular or irreducible. Strings with this property are important to study because if we can determine their breakdown combinatorially, rather than by explicitly searching through the string for runs, this will open the possibility for an $O(n)$ time algorithm for finding all runs in a string without the heavy preprocessing as required in Runs I and II. We will then proceed by stating conjectures which calculate the periodicity of these strings without the need for brute force analyses and present proofs to some of them. The exact conditions posed on these three squares are given in Lemma 2.0.7 and the resulting structure displayed in (2.2). As shown in (Fan *et al.*, 2006), such a layout is possible under 14 well defined subcases represented in Table 4.1, and every satisfying string generated by the algorithm will belong to one of these subcases. The following material follows along that of (Kopylov and Smyth, 2010).

The function ***construct_x*** in Figure 4.1 outlines the main algorithm of the program. The body

Table 4.1: The 14 subcases identified in Fan *et al.* (2006), slightly modified, for three neighbouring squares $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$ (with $v-u < w < v$, $w \neq u$).

Subcase S	k	$k+w$	$k+2w$	Special Conditions
1	$0 \leq k \leq u_1$	$k+w \leq u$	$k+2w \leq u+u_1$	$k \geq u_2$
2	$0 \leq k \leq u_1$	$k+w \leq u$	$k+2w \leq u+u_1$	$k < u_2$
3	$0 \leq k \leq u_1$	$k+w \leq u$	$k+2w > u+u_1$	—
4	$0 \leq k \leq u_1$	$u < k+w \leq u+u_1$	—	—
5	$0 \leq k \leq u_1$	$u+u_1 < k+w \leq v$	—	—
6	$0 \leq k \leq u_1$	$v < k+w < 2u$	—	—
7	$u_1 < k < u_1+u_2$	$k+w \leq u+u_1$	$k+2w \leq 2u$	—
8	$u_1 < k < u_1+u_2$	$k+w \leq u+u_1$	$k+2w > 2u$	—
9	$u_1 < k < u_1+u_2$	$u+u_1 < k+w \leq v$	—	$w < u$
10	$u_1 < k < u_1+u_2$	$k+w \leq v$	$k+2w \leq u+v$	$w > u$
11	$u_1 < k < u_1+u_2$	$k+w \leq v$	$u+v < k+2w \leq 2v-u_2$	—
12	$u_1 < k < u_1+u_2$	$k+w \leq v$	$2v-u_2 < k+2w$	—
13	$u_1 < k < u_1+u_2$	$v < k+w \leq 2u$	—	—
14	$u_1 < k < u_1+u_2$	$2u < k+w < 2u+u_2-1$	—	—

is structured as four nested ***for*** loops which when given two values u_1_max and u_2_max, iterate over all instances of $u = 2u_1 + u_2, v = 3u_1 + 2u_2, k$ and w such that for every $u_1 \in 1 \ldots u_1\_max$, $u_2 \in 1 \ldots u_2\_max$, $k \in 0 \ldots u_1 + u_2 - 1$, $w \in u_1 + u_2 + 1 \ldots v - 1$, $w \neq u$ (lines 1-5). As each string is generated, the method ***compute_subcase*** trivially sorts it into the appropriate subcase S.

Initially the maximum alphabet size of $\boldsymbol{x}$ is $\sigma_0 = u_1 + u_2$, since by (2.2) $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$ are assembled from substrings $\boldsymbol{u_1}$ and $\boldsymbol{u_2}$. Let $\boldsymbol{u^*} = \boldsymbol{u_1 u_2}$ and let the initial alphabet $\Sigma_0 = \{1, 2, \ldots, u^*\}$ with

$$\boldsymbol{u_1} = 1, 2, \ldots, u_1 \quad \textit{and} \quad \boldsymbol{u_2} = u_1 + 1, u_1 + 2, \ldots, u^*.$$

To determine the alphabet size σ, we introduce the existence of $\boldsymbol{w^2}$ by applying the condition

$$\boldsymbol{x}[k+1 \ldots k+w] = \boldsymbol{x}[k+w+1 \ldots k+2w], \tag{4.1}$$

where at each position $i \in 1..w$ in $\boldsymbol{w}$ such that $\boldsymbol{x}[k+i] = \boldsymbol{u^*}[j_1]$ for some $j_1 \in 1..u^*$, and such that

$x[k+w+i] = u^*[j_2]$ for some $j_2 \in 1..u^*$,[1] we require that the letter $u^*[j_1]$ equal the letter $u^*[j_2]$. If these letters are not already equal, then in every copy of u_1 or u_2 in x, we replace the numerically larger of the two by the smaller, updating the alphabet at each step as follows:

$$\Sigma \leftarrow \Sigma - \{\max(u^*[j_1], u^*[j_2])\}, \tag{4.2}$$

where initially $\Sigma = \Sigma_0$. After all w such pairs of positions have been considered, the letters remaining in Σ are exactly those that occur in x: $\sigma = |\Sigma|$. This is true because given $w_{min} = u_1 + u_2 + 1$, then $2w_{min} = u + u_1 + 2$ and hence $\forall k \in \{0 \ldots u_1 + u_1 - 1\}$ w^2 must span all $u_1 + u_2$ characters in the alphabet. Figures 4.1 and 4.2 outline these calculations and Figure 4.3 gives an example of an output string with characteristics $X = \{u_1 = 2, u_2 = 4, k = 1, w = 12\}$.

```
— For every subcase (u1, u2, k, w) determined by u1_max, u2_max,
— compute subcase identifier S, maximum alphabet σ and u1, u2
1.  for u1 ← 1 to u1_max do
2.    for u2 ← 1 to u2_max do
3.      for k ← 0 to u1 + u2 − 1 do
4.        for w ← u1 + u2 + 1 to v−1 do
5.          if w ≠ 2u1 + u2 then
6.            σ ← u1 + u2
7.            u1 = 1, 2, ..., u1; u2 = u1 + 1, u1 + 2, ..., u*
8.            (σ, u1, u2) ← force_square(σ, u1, u2, k, w)
9.            S ← compute_subcase  — from Table 4.1
10.           return (S, σ, u1, u2, k, w)
```

Figure 4.1: Algorithm ***construct_x***$(S, \sigma, u_1, u_2, k, w)$: $u_1 \in 1..u_1\_max$, $u_2 \in 1..u_2\_max$

```
function force_square(σ, u1, u2, k, w)
— For given values k and w, apply condition (4.1) to
— recompute σ, u1, u2 in x = u1u2u1u1u2u1u2u1u1u2
1. wlim ← min(k + w, x−w)  — possibly k + 2w > x
2. for i ← k + 1 to wlim do
3.   if x[i] ≠ x[i + w] then
4.     σ ← σ−1
5.     replace all occurrences of max(x[i],x[i + w]) in x
       with min(x[i],x[i + w])
6. return (σ, u1, u2)
```

Figure 4.2: Function ***force_square***(σ, u_1, u_2, k, w)

[1] In subcases 13 and 14 it may happen that $k + w + i > 2v$; for such values of i, therefore, no such j_2 exists.

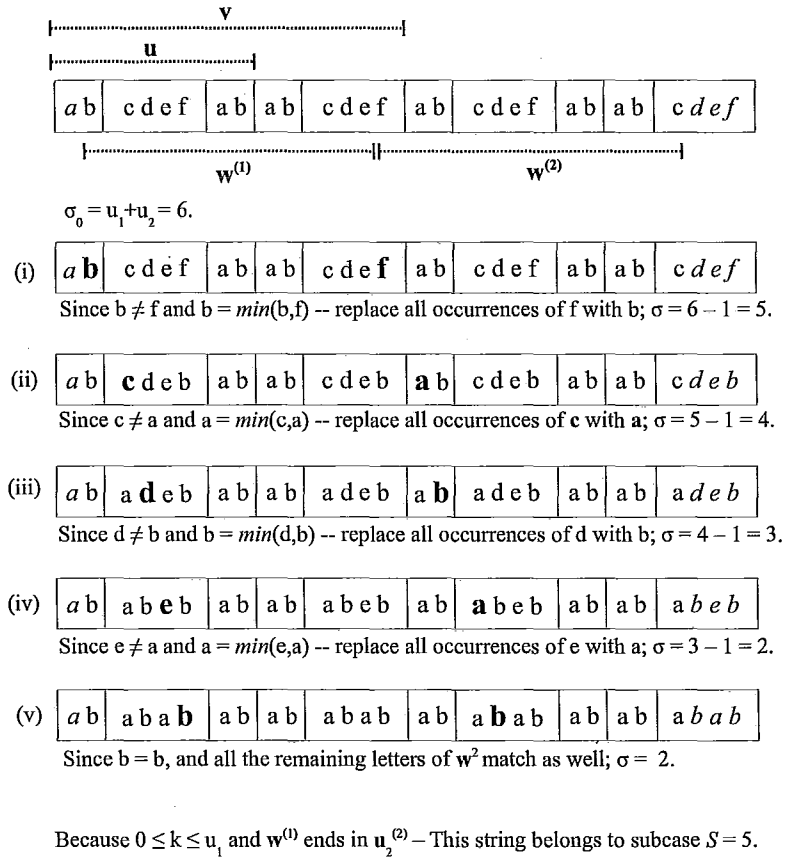

Figure 4.3: Generated string x for $X = \{u_1 = 2, u_2 = 4, k = 1, w = 12\}$.

This software is the first of its kind and was created foremost to aid in generating conjectures, hence improving its runtime or storage space is not of immediate importance. However, we will briefly touch upon both for a clearer understanding of the implementation scheme.

4.1.1 Time Complexity

Over all k and w, the time is $O(wu^2(2v-3))$ for each choice of u_1 and u_2.

Hence, the time over all values of u_1 and u_2 is

$$O(u_1\_max * u_2\_max * wu^2(2v-3)). \tag{4.3}$$

The complete details of these calculations are provided in the Appendix under 'Calculating Asymptotic Complexity'.

4.1.2 Space Complexity

The algorithm creates and processes one string at a time, deleting all of its information except the statistics required for Table 4.2 prior to generating a new string. Therefore, most of the implementation can be done in $O(x)$ bytes of space.

4.2 Generated Conjectures

Algorithm ***construct_x*** was executed for $u_1\_max = u_2\_max = 30$, yielding a total of $1,415,925$ strings spread over the 14 cases as shown in Table 4.2. In this table,

* column 2 gives the number of strings generated for Subcase S;
* σ_{max} is the maximum over all maximum alphabet sizes σ computed for any string generated for Subcase S;
* $d = \gcd(u_1, u_2, w)$ and columns 4 and 5 count the number of generated strings for which σ equals or exceeds d, respectively;
* column 6 gives the number of generated strings for which the alphabet resulting from function ***force_square*** consists of consecutive integers $1, 2, \ldots, \sigma$.
* column 7 gives the number of generated strings for which there exists a gap in the alphabet, for example a string x with $\Sigma = \{1, 2, 4\}$ is defined on a gapped alphabet.

A computer-based analysis of the generated strings counted in Table 4.2 yields a collection of conjectures, summarized in Table 4.3. We have discovered that whenever $\sigma = d$, x breaks down into a repetition of period d. If $\sigma > d$, the string is not fully repetitive but does show a highly periodic behaviour. For Subcases $3, 4, 7$, the breakdown depends on parameters

$$s = \gcd(u - w, w - u_1);\ \alpha = \lfloor u/s \rfloor;\ \gamma = \lfloor v/s \rfloor;\ \epsilon = (u_1 + u_2)/s; \tag{4.4}$$

Table 4.2: Statistics for 1,415,925 strings generated using $u_1\_max = u_2\_max = 30$

1	2	3	4	5	6	7
S	# strings	σ_{max}	# $\sigma = d$	# $\sigma > d$	# $\Sigma = \{1, 2, \ldots, \sigma\}$	# gaps
1	7840	7	7840	0	7840	0
2	8960	10	8960	0	8960	0
3	131100	29	118305	12795	131100	0
4	283620	30	276799	6821	278132	5488
5	227505	30	227505	0	227505	0
6	121800	15	121800	0	121800	0
7	47250	27	44548	2702	44860	2390
8	51640	15	51640	0	51640	0
9	90335	15	90335	0	90335	0
10	64050	10	64050	0	64050	0
11	54000	15	51707	2293	54000	0
12	16800	15	15612	1188	16800	0
13	201405	30	197860	3545	201405	0
14	109620	15	108770	850	108831	789

while for Subcases 11–14, it depends on

$$t = v - w; \beta = \lfloor 2u/t \rfloor - 1. \tag{4.5}$$

Table 4.3: Overview of Conjectures

Subcases S	Conditions	Breakdown of $\boldsymbol{x}/\boldsymbol{v}^2$
1,2,5,6,8–10	$(\forall \boldsymbol{x}, \sigma = d)$	$\boldsymbol{x} = \boldsymbol{d}^{(x/d)}$
3,4,7	$\sigma = d$	$\boldsymbol{x} = \boldsymbol{d}^{(x/d)}$
	$\sigma > d$	$\boldsymbol{x} = \boldsymbol{s}^{\alpha}\boldsymbol{s}[1 \ldots u_1 \bmod s]\boldsymbol{s}^{\gamma}\boldsymbol{s}[1 \ldots u_1 \bmod s]\boldsymbol{s}^{\epsilon}$
11–14	$\sigma = d$	$\boldsymbol{x} = \boldsymbol{d}^{(x/d)}$
	$\sigma > d$	$\boldsymbol{v}^2 = (\boldsymbol{t}^{\beta}\boldsymbol{t}[1..t \bmod u_1])^2$

The conjectures for subcases 1,2,5,6,8–10 in Table 4.3 have been proven correct in (Kopylov and Smyth, 2010) and the resulting lemma stated in Chapter 5. The proofs of these subcases were primarily carried out by Bill Smyth, however Tim Paterson and I have participated in supplementing them with ideas and corrections. Since it is known that for subcases 1,2,5,6,8-10 $\boldsymbol{x}$ is a repetition of period $d = \gcd(u_1, u_2, w)$, then $\sigma \not> d$ or otherwise such a repetition may not always exist. Moreover,

since a repetition of period d can always be represented using d distinct letters, it follows that $\sigma \not< d$. In other words, $\sigma = d$ is a condition necessary for periodicity d. The first clues to the source of this statement were found by string analyses and are shown in columns 4 and 5 of Table 4.2, where all generated strings for these cases have $\sigma = d$. However, this condition is not fully sufficient for Subcases 3, 4, 7, 11–14, and the complete conjectures are dependent on further parameters. Note that according to the experiments done so far, about 96% of the generated strings $\boldsymbol{x}$ yield $\sigma = d$ and so reduce to $\boldsymbol{x} = \boldsymbol{d}^{(x/d)}$. Furthermore, out of 1,415,925 strings only 8667 (about 0.6%) have a prefix of length σ in which some letter is necessarily duplicated, for example the string

$$\boldsymbol{x} = (aac)^3 a (aac)^5 a (aac)^2 \tag{4.6}$$

in subcase 4. These are combinatorial remarks which have yet to be pursued. Below are a few examples of strings from each collection of subcases in Table 4.3.

Example 1: $X = \{u_1 = 12, u_2 = 8, k = 10, w = 36, d = 4\}$ - subcase 5

Based on the conjecture in Table 4.3, the string $\boldsymbol{x}$ with these characteristics must break down into $\boldsymbol{x} = \boldsymbol{d}^{(x/d)}$. Then it follows that $d = \gcd(u_1, u_2, w) = \gcd(12, 8, 36) = 4$. Therefore, $\boldsymbol{x}$ is a repetition of 4 distinct characters, $x/d = 104/4 = 26$ times. Providing these conditions to the software as input, it generates the string $\boldsymbol{x} = (abcd)^{26}$. □

Example 2: $X = \{u_1 = 4, u_2 = 2, k = 4, w = 7, d = 1\}$ - subcase 4

In this example, we will demonstrate how the string (4.6) is determined using the parameters (4.4). Firstly $u = 10$, $v = 16$ and $x = 32$. Next,

$$s = \gcd(3, 3) = 3; \quad \alpha = \lfloor \frac{10}{3} \rfloor = 3; \quad \gamma = \lfloor \frac{16}{3} \rfloor = 5; \quad \epsilon = \frac{6}{3} = 2$$

and $\boldsymbol{s}[1 \ldots u_1 \bmod s] = \boldsymbol{s}[1 \ldots 1]$. Based on the conjecture in Table 4.3, the string $\boldsymbol{x}$ with these characteristics must break down into

$$\boldsymbol{x} = \boldsymbol{s}^{\alpha} \boldsymbol{s}[1 \ldots u_1 \bmod s] \boldsymbol{s}^{\gamma} \boldsymbol{s}[1 \ldots u_1 \bmod s] \boldsymbol{s}^{\epsilon}.$$

From our calculated parameters, $\boldsymbol{x}$ is formed through multiple runs of the substring $\boldsymbol{s}$, or

$$\boldsymbol{x} = \boldsymbol{s}^3\boldsymbol{s}[1]\boldsymbol{s}^5\boldsymbol{s}[1]\boldsymbol{s}^2 \quad with \;\; s = 3 \tag{4.7}$$

The software generated string is (4.6) and our combinatorially generated string is (4.7). Other than precisely knowing the alphabet [2] of $\boldsymbol{x}$, these two strings are equivalent in their structure. □

Example 3: $X = \{u_1 = 11, u_2 = 8, k = 13, w = 34, d = 1\}$ - subcase 11

Firstly $u = 30$, $v = 49$ and $x = 98$. Next,

$$t = v - w = 49 - 34 = 15; \quad \beta = \left\lfloor \frac{60}{15} \right\rfloor - 1 = 3$$

and $\boldsymbol{t}[1 \ldots t \bmod u_1] = \boldsymbol{t}[1 \ldots 4]$. Based on the conjecture in Table 4.3, the string $\boldsymbol{x}$ with these characteristics must break down into the form

$$\boldsymbol{x} = (\boldsymbol{t}^\beta \boldsymbol{t}[1..t \bmod u_1])^2.$$

From our calculated parameters, $\boldsymbol{x}$ is formed through multiple runs of the substring $\boldsymbol{t}$, or

$$\boldsymbol{x} = \boldsymbol{t}^3\boldsymbol{t}[1 \ldots 4]\boldsymbol{t}^3\boldsymbol{t}[1 \ldots 4] \quad with \;\; t = 15 \tag{4.8}$$

The software generated string is

$$\boldsymbol{x} = (abbbabbbabbbabb)^3 abbb (abbbabbbabbbabb)^3 abbb \tag{4.9}$$

In this case, the substring $\boldsymbol{t}$ consists of runs itself; however the exact combinatorial form has yet to be deduced. □

[2] Determining the definite sequence of letters in $\boldsymbol{x}$ is a question for further research and is discussed in Chapter 5.

Chapter 5

New Combinatorial Results Based on Conjectures

5.0.1 Subcases 1, 2, 5, 6, 8-10

Our main result is the following Lemma 5.0.1 which confirms the conjectures for seven of the 14 subcases identified in Table 4.3. The detailed proofs may be found in (Kopylov and Smyth, 2010). Recall from the discussion at the end of Chapter 2 that in this lemma the regularity condition on u^2 is no longer required. We suppose of course that all squares are irreducible.

Lemma 5.0.1 (Figures 5.1-5.5, Subcases 5, 6, 8, 9 & 10). *If*

(*a*) $0 \leq k \leq u_1$ *and* $u + u_1 < k + w \leq v$, **or**

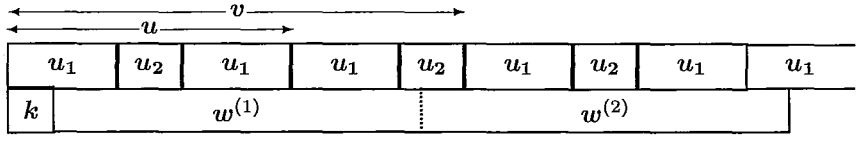

Figure 5.1: (a) Subcase 5

(*b*) $0 \leq k \leq u_1$ *and* $v < k + w \leq 2u$, **or**

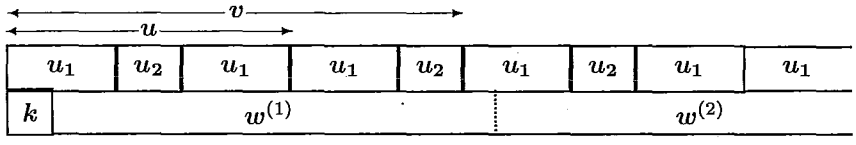

Figure 5.2: (b) Subcase 6

(*c*) $u_1 < k < u_1 + u_2$ *and* $u < k + w \leq u + u_1$ *and* $2u < k + 2w$, **or**

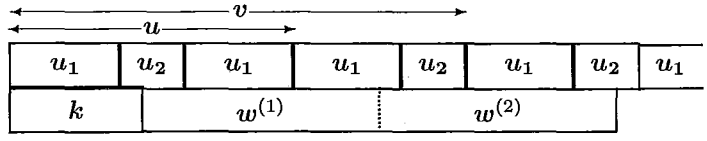

Figure 5.3: (c) Subcase 8

(*d*) $u_1 < k < u_1 + u_2$ *and* $u + u_1 < k + w \leq v$ *and* $w < u$, **or**

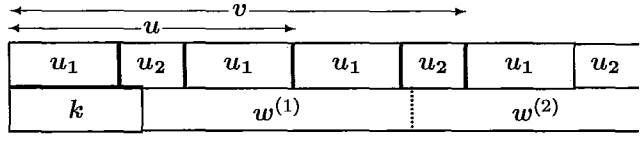

Figure 5.4: (d) Subcase 9

(e) $u_1 < k < u_1 + u_2$ *and* $k + w \leq v$ *and* $k + 2w \leq u + v$ *and* $w > u$,

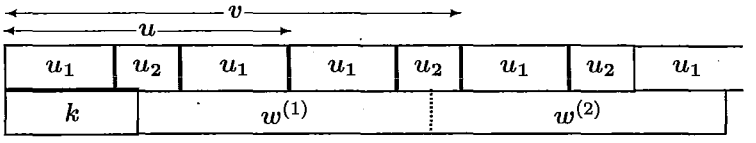

Figure 5.5: (e) Subcase 10

then v^2 *is a repetition of period* $\gcd(u_1, u_2, w)$.

This lemma captures precisely the combinatorial consequence of having three squares occurring at neighbouring positions in a string. It provides restrictions on the possible periodic substrings and gives definite information about the occurrences of repetitions in x – both properties not fully considered by Lemmas 2.0.3 and 2.0.4. Interestingly, though the combinatorial result for each of the 7 subcases is the same, the strategic approach for each proof varies significantly.

5.0.2 Remaining Subcases and Other Mysteries

The unproved conjectures for subcases 3,4,7,11-14 in Table 4.3 provide guidance for future research. Perhaps for all remaining cases, the assumption for which $\sigma = d$ should be dealt with first because of the simple repetitive breakdown it exhibits. However, proving the conjectures with $\sigma > d$ will allow for the number runs to be easily computed, as demonstrated in 4.8, rather than directly searched for in a string.

Further combinatorial mysteries arising from our research are the following:

* What is the upper bound on the alphabet size σ? By observation, it appears that $\sigma \leq (u_1 + u_2)/2$.

* Does $\sigma = \gcd(u_1, u_2, w)$ imply that x is a repetition of period d? If so, can this fact be used to simplify proofs?

* Let x be a string generated by the function ***force_square***. If it includes a maximum letter greater than alphabet size σ, does it follow that $\sigma > \gcd(u_1, u_2, w)$?

* What is the combinatorial breakdown for subcases 13-14 where $\sigma > d$ and $k + 2w > v^2$?

* How do we determine the sequence of symbols in substrings s and t for subcases where $\sigma > d$?

Along with Tim Paterson, we have started to examine Subcase 3, illustrated in Figure 5.6.

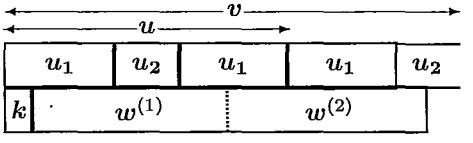

Figure 5.6: Subcase 3

For strings with $\sigma > d$, evidence shows that the substring s forming the conjectured breakdown

$$x = s^{\alpha} s[1 \ldots u_1 \bmod s] s^{\gamma} s[1 \ldots u_1 \bmod s] s^{\epsilon}$$

may be recovered by using information on its length s and the parameter

$$l = u_1 \bmod s \tag{5.1}$$

conjectured to represent the length of its longest border.

The following example illustrates how to recover the substring s, and thus the entire string x, for subcase 3 using the conjectured parameters (4.4) and (5.1):

Example 4: $X = \{u_1 = 4, u_2 = 2, k = 2, w = 7, d = 1\}$ - subcase 3

Firstly $u = 10$, $v = 16$ and $x = 32$. Next,

$$s = \gcd(3,3) = 3; \quad \alpha = \lfloor \frac{10}{3} \rfloor = 3; \quad \gamma = \lfloor \frac{16}{3} \rfloor = 5; \quad \epsilon = \frac{6}{3} = 2; \quad l = 4 \bmod 3 = 1$$

and $s[1 \ldots u_1 \bmod s] = s[1 \ldots 1]$.

Based on the conjecture in Table 4.3, the string x with characteristics X must break down into the form

$$x = s^{\alpha} s[1 \ldots u_1 \bmod s] s^{\gamma} s[1 \ldots u_1 \bmod s] s^{\epsilon}. \tag{5.2}$$

From our calculated parameters, x is formed through multiple runs of the substring s, or

$$x = s^3 s[1] s^5 s[1] s^2$$

Next, since $s = 3$ and the longest border $l = 1$, then the only possibility for its sequence of letters is $s = aba$. Therefore, the resulting string x must be:

$$x = (aba)^3 a (aba)^5 a (aba)^2. \tag{5.3}$$

Agreeably, the software generated string is

$$x = (aba)^3 a (aba)^5 a (aba)^2. \qquad \square$$

Computer analysis has shown that the paramater (5.1) consistently matches the length of the longest border of s for all strings in subcase 3. Proving that l indeed represents the length of the longest border of s is the first step to confirming the sequence of letters in s. Moreover, an obvious question arises whether there is a connection between l and the length of the tail $s[1 \ldots u_1 \bmod s]$ in (5.2).

Further work relating to the three squares research will be periodically updated on the website

`http://www.cas.mcmaster.ca/~bill/cv.shtml`.

Appendix A

A.1 Runtime for LZ-factorization algorithms

Table A.1: Runtime in microseconds per input symbol for various LZ factorization algorithms.

String	AKO	CPS1-2	CPS1-3b	CI1	CI2	CIS	CII	CPS2	CPS3	OS
fibo36	1.91	1.51	1.56	**0.51**	1.51	1.63	1.54	0.52	**0.17**	3.85
fss10	1.86	1.48	1.58	**0.52**	1.48	1.48	1.53	0.52	**0.16**	4.11
rand2	0.84	0.48	0.56	**0.43**	0.47	0.47	0.51	1.88	**0.33**	5.69
rand21	0.64	0.51	0.59	0.53	0.49	**0.48**	0.52	2.83	**0.66**	13.86
chr22	0.84	0.54	0.65	0.56	**0.51**	0.60	0.55	2.96	**1.10**	10.20
chr19	0.90	0.59	0.72	0.60	0.57	**0.56**	0.60	3.30	**2.57**	9.89
prot-a	0.76	0.57	0.66	0.59	**0.47**	0.55	0.58	**2.73**	3.10	13.65
bible	0.60	0.40	0.48	**0.35**	0.39	0.45	0.42	**1.33**	22.19	6.09
howto	0.78	0.55	0.68	**0.47**	0.53	0.62	0.56	**1.88**	-	10.79
mozilla	0.59	0.45	0.60	0.59	**0.44**	0.51	0.47	**1.72**	-	14.16

A.2 Space for LZ-factorization algorithms

Table A.2: Peak memory usage in words per input symbol for the LZ factorization algorithms.

String	AKO	CPS1-2	CPS1-3b	CI1	CI2	CIS	CII	CPS2	CPS3	OS
fibo36	23.31	4.05	3.05	3.25	**3.00**	**3.00**	**3.00**	1.44	1.39	**1.15**
fss10	23.08	3.94	**2.93**	3.25	3.00	3.00	3.00	1.44	1.42	**1.27**
rand2	22.81	3.52	**2.52**	3.25	3.00	3.00	3.00	1.44	1.50	**1.15**
rand21	10.26	3.86	**2.86**	3.25	3.00	3.00	3.00	1.44	1.38	**1.15**
chr22	16.64	3.65	**2.65**	3.25	3.00	3.00	3.00	1.44	1.41	**1.20**
chr19	16.85	3.65	**2.65**	3.25	3.00	3.00	3.00	1.44	1.34	**1.22**
prot-a	12.11	3.82	**2.82**	3.25	3.00	3.00	3.00	1.44	1.31	**1.16**
bible	14.27	3.72	**2.27**	3.25	3.00	3.00	3.00	1.44	1.32	**1.16**
howto	14.73	3.73	**2.25**	3.25	3.00	3.00	3.00	1.44	-	**1.22**
mozilla	10.52	3.95	**2.95**	3.25	3.00	3.00	3.00	1.44	-	**1.26**

A.3 Calculating Asymptotic Complexity

We begin with,

$$T = \sum_{u_1=1}^{u_1\_max} \Big(\sum_{u_2=1}^{u_2\_max} \Big(\sum_{k=0}^{u_1+u_2-1} \Big(\sum_{w=u_1+u_2+1}^{3u_1+2u_2-1} O(xw) \Big) \Big) \Big) \tag{A.1}$$

Then,

$$\begin{aligned}
\sum_{w=u_1+u_2+1}^{3u_1+2u_2-1} O(xw) &= O(xw(3u_1+2u_2-1-u_1-u_2-1+1)) \\
&= O(xw(2u_1+u_2-1)) \\
&= O(xw(u-1)) \\
&= O(w(3u+u_2)(u-1)) \quad since \quad x = 3u+u_2 \\
&= O(w(3u^2-3u+u*u_2-u_2)) \\
&= O(w(3u^2+(u_2-3)u) \\
&= O(w(3(2u_1+u_2)^2+(u_2-3)(2u_1+u_2))) \\
&= O(w(3(4u_1^2+4u_1u_2+u_2^2)+2u_1u_2+u_2^2-3u)) \\
&= O(w(12u_1^2+14u_1u_2+4u_2^2-3u)) \\
&= O(w(2(6u_1^2+7u_1u_2+2u_2^2)-3u)) \\
&= O(w(2(3u_1+2u_2)(2u_1+u_2)-3u)) \\
&= O(w(2vu-3u)) \\
&= O(wu(2v-3))
\end{aligned}$$

Next,

$$\begin{aligned}
\sum_{k=0}^{u_1+u_2-1} O(wu(2v-3)) &= O(wu(2v-3)(u_1+u_2-1+1)) \\
&= O(wu(2v-3)(u_1+u_2)) \\
&= O(wu(2v-3)(u-u_1)) \quad since \quad u = 2u_1+u_2 \\
&= O(wu^2(2v-3))
\end{aligned}$$

And finally,

$$\sum_{u_1=1}^{u_1\_max}\Big(\sum_{u_2=1}^{u_2\_max} O(wu^2(2v-3))\Big) = O(u_1\_max * u_2\_max * wu^2(2v-3))$$

Bibliography

Abouelhoda, M., Kurtz, S., and Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *Theoretical Computer Science*, **2**, 53–86.

Al-Hafeedh, A., Crochemore, M., Ilie, L., Kopylov, J., Smyth, W., Tischler, G., and Yusufu, M. (2010). A Comparison of Lempel-Ziv LZ77 Factorization Algorithms. *ACM Computing Surveys*, **0**.

Apostolico, A. and Preparata, F. (1983). Optimal off-line detection of repetitions in a string. *Theoret. Comput. Sci.*, **22**, 297–315.

Bender, M. and Farach-Colton, M. (2000). The LCA problem revisited. *Latin American Theoretical Informatics*, pages 88–94.

Bennet, A. and Bennet, D. (2008). The human knowledge system: music and brain coherence. *VINE: The journal of information and knowledge management systems*, **38**, 277–295.

Berstel, J., Lauve, A., Reutenauer, C., and Saliola, F. (2009). *Combinatorics on Words*, volume 27. American Mathematical Society.

Bullard, B. (2003). Metamusic: Music for inner space. *Hemi-Sync Journal*, **XXI**, i–v.

Cambouropoulos, E., Crochemore, M., Iliopoulos, C. S., Mouchard, L., and Pinzon, Y. J. (1999). Algorithms for Computing Approximate Repetitions in Musical Sequences. In *International Journal of Computer Mathematics*, pages 129–144.

Chen, B., Paterson, M., and Zhang, G. (2007a). A new succinct representation of rmq-information and improvements on the enhanced suffix array. *LNCS*, **4614**, 459–470.

Chen, G., Puglisi, S. J., and Smyth, W. (2007b). Fast and practical algorithms for computing all runs in a string. *LNCS*, **4580**, 307–315.

Chen, G., Puglisi, S. J., and Smyth, W. (2008). Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science*, **1**(4), 605–623.

Crawford, T., Iliopoulos, C. S., and Raman, R. (1998). String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, **11**, 73–100.

Crochemore, M. (1981). An optimal algorithm for computing the repetitions in a word. *Inform. Process. Lett.*, **12**, 244–250.

Crochemore, M. (1984). Linear searching for a square in a word. *Bull. EATCS*, **24**, 66–72.

Crochemore, M. (1986). Transducers and repetitions. *TCS*, **45**(1), 63–86.

Crochemore, M. and Ilie, L. (2008). Computing Longest Previous Factor in linear time and applications. *Information Processing Letters*, **106**(2), 75–80.

Crochemore, M. and Rytter, W. (1995). Squares, cubes, and time-space efficient string searching. *Algorithmica*, **13**(5), 405–425.

Crochemore, M., Landau, G. M., and Ziv-Ukelson, M. (2002). A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. pages 679–688. Proc. 12th ACM-SIAM Symp. Discrete Algs.

Crochemore, M., Ilie, L., and Smyth, W. (2008). A simple algorithm for computing the Lempel-Ziv factorization. pages 482–488. Proc. 18th Data Compression Conference (DDC'08).

Fan, K., Smyth, W., Puglisi, S., and Turpin, A. (2006). A New Periodicity Lemma. *SIAM. J. Discrete Math*, **20**, 656–668.

Farach, M. (1997). Optimal suffix tree construction with large alphabets. pages 137–143.

Fine, N. and Wilf, H. (1965). Uniqueness theorems for periodic functions. *Proc. Amer. Math. Soc.*, **16**, 109–114.

Franek, F., Holub, J., Smyth, W., and Xiao, X. (2003). Computing quasi suffix arrays. *J. Automata, Languages & Combinatorics*, **8**(4), 593–606.

Harel, D. and Tarjan, R. (1984). Fast algorithms for finding nearest common ancestors. *SIAM J. Computing*, **13**(2), 338–355.

Hopcroft, J. (1971). An nlogn algorithm for minimizing states in a finite automaton. *Theory of Machines and Computations, Academic Press*, pages 189–196.

Ilie, L., Crochemore, M., and Tinta, L. (2008). Towards a solution to the 'runs' conjecture. *Lecture Notes in Computer Science*, **5029**, 290–302.

Kärkkäinen, J. and Sanders, P. (2003). Simple linear work suffix array construction. *LNCS*, **2719**, 943–955.

Kärkkäinen, J., Manzini, G., and Puglisi, S. J. (2009). Permuted longest-common-prefix array. *LNCS*, **5577**, 181–192.

Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K. (2001). Linear-time longest-common-prefix computation in suffix arrays and its applications. *LNCS*, **2089**, 181–192.

Ko, P. and Aluru, S. (2003). Space efficient linear time construction of suffix arrays. *LNCS*, **2676**, 200–210.

Kolpakov, R. and Kucherov, G. (1998). Maximal Repetitions in Words, or How to Find All Squares in Linear Time. Rapport Interne LORIA 98-R-227, Laboratoire Lorrain de Recherche en Informatique et ses Applications.

Kolpakov, R. and Kucherov, G. (1999). Finding maximal repetitions in a word in linear time. pages 596–604. Proc. 40th Annual IEEE Symp. Found. Computer Science.

Kolpakov, R. and Kucherov, G. (2000). On maximal repetitions in words. *J. Discrete Algorithms*, **1**, 159–186.

Kopylov, E. and Smyth, W. (2010). The Three Squares Lemma Revisited. *J. Discrete Algorithms (accepted subject to revision).*

Lee, S. and Park, K. (2007). Dynamic rank-select structures with applications to run-length encoded texts. *LNCS*, **4580**, 95–106.

Lempel, A. and Ziv, J. (1976). On the complexity of finite sequences. *IEEE Trans. Informative Theory*, **22**, 75–81.

Lempel, A. and Ziv, J. (1977). A universal algorithm for sequential data compression. *IEEE Trans. Informative Theory*, **23**, 337–343.

Lempel, A. and Ziv, J. (1978). Compression of individual sequences via variable-rate coding. *IEEE Trans. Informative Theory*, **24**, 530–536.

Main, M. (1989). Detecting leftmost maximal periodicities. *Discrete Appl. Math.*, **25**, 145–153.

Main, M. and Lorentz, R. (1984). An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. Algorithms*, **5**, 422–432.

Maniscalco, M. and Puglisi, S. J. (2006). Faster lightweight suffix array construction. pages 16–29. Proc. 17th AWOCA.

Manzini, G. (2004). Two space saving tricks for linear time LCP computation. *LNCS*, **3111**, 372–383.

Manzini, G. and Ferragina, P. (2004). Engineering a lightweight suffix array construction algorithm. *Algorithmica*, **40**, 33–50.

McCreight, E. (1976). A space-economical suffix tree construction algorithm. *J. ACM*, **23**, 262–272.

Nong, G., Zhang, S., and Chan, W. H. (2009). Linear time suffix array construction using D-critical substrings. *LNCS*, **5577**, 54–67.

Pop, M. (2009). Genome assembly reborn: recent computational challenges. *Briefings in Bioinformatics*, **10**(4), 354–366.

Puglisi, S. and Simpson, J. (2008). The expected number of runs in a word. *Australasian Journal of Combinatorics*, **42**, 45–54.

Puglisi, S. J. and Turpin, A. (2008). Space-time tradeoffs for longest-common-prefix array computation. pages 124–135. Proc. 19th ISAAC.

Puglisi, S. J., Smyth, W., and Turpin], A. (2007). A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, **39**(4), 1–31.

Schatz, M. C., Phillippy, A. M., and Pop, M. (2008). Genome assembly forensics: finding the elusive mis-assembly. *Genome Biology*, **9**(3).

Simpson, J. (2010). Modified Padovan words and the maximum number of runs in a word. *Australasian Journal of Combinatorics*, **46**, 129–145.

Smyth, W. F. (2003). *Computing Patterns in Strings.* Pearson Education Limited.

The Monroe Institute (1985). Achieving optimal learning states. *Breakthrough.*

Thue, A. (1906). Über unendliche Ziechernreihen. *Kra. Vidensk. Selsk. Skrifter. I. Mat.-Nat. Kl.*, (7).

Thue, A. (1910). Die Lösung eines Spezialfalles eines generellen logischen Problems. *Kra. Vidensk. Selsk. Skrifter. I. Mat.-Nat. Kl.*, (8).

Thue, A. (1912). Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen. *Kra. Vidensk. Selsk. Skrifter. I. Mat.-Nat. Kl.*, (10).

Thue, A. (1914). Probleme über Veränderungen von Zeichenreihen nach gegebenen Regeln. *Kra. Vidensk. Selsk. Skrifter. I. Mat.-Nat. Kl.*, (7).

Ukkonen, E. (1992). Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, **92**(1), 191–211.

Weiner, P. (1973). Linear pattern matching algorithms. In *14th Annual IEEE Symp. Switching & Automata Theory*, pages 1–11.